Process-Level, Time-Driven, Simulation of a Computer Network on a Parallel Shared-Memory Processor

B.D. Lubachevsky and K.G. Ramakrishn[1]

Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

Ultracomputer Note #88

October, 1983

ABSTRACT

This paper reports on a parallel, time-driven simulation of a computer network designed to run on a shared-memory MIMD multiprocessor. The NYU Ultracomputer is chosen as the model of a shared-memory MIMD machine. Various results on efficiency and speedup are reported. It may be inferred from the results presented in the report, that a time-driven simulation, though inefficient in the uniprocessor context, can prove to be a useful tool in the parallel context. A major part of this paper is devoted to algorithms for coordination and synchronization of processors in the ultracomputer. A dynamic task manager for scheduling tasks is also discussed. Actual FORTRAN codes are presented for the synchronization and scheduling of processors. A brief overview of a new table-driven approach for simulating computer networks at process-level, is also presented. This method has the potential for enormous savings in development time of simulation programs. After a discussion of results on speedup and efficiency of our parallel simulator, a simple theoretical model is developed to predict the efficiency for larger computer networks. This method takes into account the synchronization overhead and concurrency of events.

1. Introduction

Although serial *event driven* (ED) simulation has proven to be both a general and a practical method [3, 9], its parallel counterpart has not yet yielded efficient tools for large enough class of problems. The serial *time driven* (TD) simulation is indisputably poorer than serial ED simulation. The imminent availability of multiprocessors, resurrects, we believe, both a

[1]This work was supported by DOE grant number AC0276ER03077.

theoretical and practical interest in TD simulation of a large class of physical systems that possess an inherent synchronism. However, not too many reports on the parallel TD simulation has been recently seen.

In this paper, we develop a simulator of a computer network consisting of tens of heterogeneous processors servicing hundreds of on-line inquiry/response terminals. There are two essential assumptions about the network which make its parallel TD simulation feasible: (I) inherent synchronism of the whole network, (II) high density of concurrent events throughout the network. Item (I) is induced by the time slice mechanism which is common to all processors in the network and tightly coupled communication interface. The process level of our simulation, made possible by our table-driven mechanism, induces item (II). Being a paradigm of a class of computer networks, the analysis of this example, we believe, will help develop practical simulation tools.

The parallel host computer we chose for the simulator is a shared memory asynchronous processor, enhanced with the basic primitive *Fetch&Add*. The MIMD shared memory architecture presents the following main advantages in comparison to message passing architecture: efficiency of the solutions for a large class of problems (not only for those networks, for example, whose geometry agree with the geometry of the message passing machine), more flexible program structure allowing dynamic network reallocation during the execution, ease of handling global statistics, and better manageability of programming.

Since the general availability of the parallel hardware is imminent [5, 7, 12], a simulator of a promising multiprocessor design, the Ultracomputer, was chosen. This simulator truthfully emulates the environment of the Ultracomputer at the level of machine instructions. The simulator also enables one to write parallel programs in FORTRAN. Hence, the process-level, time-driven simulator was written in FORTRAN and various performance measures of the program were predicted by running the program on the simulator of the Ultracomputer. We believe these measures to qualitatively reflect the performance of our parallel program on the Ultracomputer.

In this "almost" real environment of the simulation a "potpourri" of various computer science and programming "ingredients" was unavoidable, each of which might possibly deserve a separate paper. In fact we discuss a model of parallel processor in sec. 2, give an overview of the discrete event simulation methodology in sec. 3, describe in detail a particular computer system, as well as the table-driven approach to simulating computer networks at process level in sec. 4. We also cannot avoid detailed discussion of various data structures in sec. 4 and 5 and present parallel programming codes. We analyze the experimental results of the parallel simulation in sec. 7 and comment on a way of relaxing assumption (II) in sec. 8.

To avoid getting bogged down in the quagmire of our potpourri, we hope that the following description might be of help. There are three layers in our simulation (Fig. 1). The computer network, called *target* is simulated by a parallel shared memory MIMD processor, called *host*, whose simulator in turn runs on a serial processor, called *host-simulator*. We hope it is clearly understood that this three-layers simulation is not an efficient way to simulate the original target network.

2. The model of the parallel processor

We first describe the ideal model, dubbed a "paracomputer" [12]. The paracomputer consists of *N* identical PEs sharing a common memory. The individual PEs may also have attached local memory, which we refer to as their "private" memories; the memory shared by and common to all processors is called "shared memory," and variables stored there are called "shared variables."

This original definition [12] implies a very large number *N* (thousands and more). We will consider moderate values of *N* as well. We also relax the original requirement that the PEs be identical. In our model, though the PEs contain the same instruction set, they may have different execution speeds. In fact, none of the software constructs described below makes explicit use of the fact that the instruction repertoire is identical for all PEs.

Paracomputer presumes no contention for accessing shared memory. In the case of identical PEs, this assumption may be also expressed as follows. The PEs can simultaneously read any shared cell in one cycle. Moreover, simultaneous writes, including the *Fetch&Add* operations described below, are likewise effected in a single cycle. A memory cell to which such writes are directed will contain some one of the quantities written into it. Note that simultaneous memory updates are *not* serialized; in fact they are accomplished in one cycle.

Paracomputers must be regarded as idealized computational models since physical fan-in limitations prevent their realization. One realizable approximation to a paracomputer is an "Ultracomputer" [5] in which each PE can directly access its private memory and can access the shared memory via a (multicycle) interconnection network. In this more realistic architecture a shared memory access may require several PE cycles. For moderate values of *N* simpler solutions of the problem of connecting the PEs with the shared memory are feasible, such as the crossbar switch.

The code of a parallel program looks like the one for an ordinary serial program. Each PE "sees" the shared memory and executes the program code. In this paper, the only case considered is one in which the code is identical for all the PEs. *Fetch&Add* is the primitive to coordinate PEs.

The format of the *F&A* operation is *F&A* (V,e), where *V* is a variable and *e* is an expression. This indivisible operation yields the *old* contents of *V* as its value and replaces the contents of V by $V+e$.

The following principle of serialization of operations *F&A* takes place: Assume that several (possibly very many) *F&A* operations simultaneously address V. Then the effect is as if these operations occurred in some (unspecified) serial order, i.e. V receives the appropriate total increment and each operation yields the intermediate value of V corresponding to its position in this order.

For example, if PE1 executes

$$p_1 \leftarrow F\&A\ (\mathrm{V},\mathrm{e}_1),$$

and if PE2 simultaneously executes

$$p_2 \leftarrow F\&A\ (\mathrm{V},\mathrm{e}_2),$$

and if V is not simultaneously updated by another PEk ($k \neq 1,2$), then either

$$p_1 \leftarrow V, p_2 \leftarrow V+e_1,$$

or

$$p_1 \leftarrow V+e_2, p_2 \leftarrow V,$$

and, in either case, the value of V becomes $V+e_1+e_2$. The first possibility corresponds to the serialized order in which PE1 executes its *F&A* and then PE2 executes its *F&A*; the second possibility corresponds to the opposite serialization.

It is also possible to have *load*s, *store*s, and *F&A*s all concurrently directed at the same memory location. The effect is as though these operations occurred in some serial order.

In [6,5] a hardware design is presented in which the *F&A* operation requires essentially the same execution time as a *load* or *store* and in which many simultaneous *F&A*s updating the same variable are processed particularly efficiently.

3. An overview of two methods of discrete event simulation

Simulation of a discrete event stream in a distributed system (of which network simulation is a special case) may employ the following two methods, or a mixture of them.

In the (fixed increment) TD simulation the state of the simulated system is observed before the end of each time interval of a fixed length Δt. This state is evaluated as a function of the state before the end of the previous time interval. To provide adequate simulation Δt must be of the order of the smallest time interval between two events in which the earlier event affects the later event. This method however may become inefficient, when state changes follow irregularly, since many blank computational cycles may be involved.

In the variable increment ED simulation the state of the simulated system is observed only at time instances when this state changes, i.e. when some events are happening in the system. Programming of the ED simulation is complicated even on a uniprocessor. A distributed ED simulation has been discussed in a number of papers and various ideas have been suggested. Despite this, the authors could not find an existing method, both efficient and general, to simulate a computer network.

We choose a TD simulation. To justify our choice we will give now a brief overview of the difficulties of a distributed ED simulation.

In an ED simulation, instead of explicitly maintaining the whole state of the simulated system, the pool Π of estimates of the *events* possible in future is maintained. An event (estimate) in Π is called *resumable* if it can be affected by no other event in Π.

There is always at least one resumable event in Π, namely, that with a minimal time. In the serial ED simulation this is the event to be worked on next. Parallelization can be beneficial only in the case when there are many resumable events. Each of them then presents an opportunity to create a concurrent computational process.

If the network graph is small, tightly connected and propagation delays are small, few events are usually resumable and parallelization of ED simulation for such a system seems infeasible. Note that in special cases, for example in pipe-line networks, parallelization may still be feasible even for small graphs. Yet this does not contradict the previous statement since the propagation delay is large ($+\infty$) in the counter-traffic direction. Also note that the problem of an inexpensive determination of resumable events exists even in the case when the parallelization is feasible. The suggested solutions (using only information local to a node to determine a resumable event [1], empty messages [2, 11], "rolling back" [8]) create a number of problems such as inefficiency, deadlocks, "artifact" overflow (i.e. the overflow which happens only in the host system, not in the target) etc.

We have taken an example (see sec. 4) which is a relatively small and tightly coupled network. It has many cycles since all lines are full duplex and the links of the graph are bidirectional. The round-robin cycles of all the node-processors are identical.

Faced with the problems of an ED simulation of this particular network we decided to use the time driven method for it. Although blank cycles are possible we are hoping that our network is loaded enough so that in an average round-robin cycle an essential quota of the simulated node-processors are changing their state thus justifying explicit simulation of this cycle (cf. assumption (II) in the introduction). Thus we accept the length of the round-robin cycle as Δt.

Although we simplify our task by considering time driven simulation we hope to devote one of the subsequent reports to a more detailed consideration of event driven simulation.

4. Simulated system and serial algorithm of its simulation

First, in sec. 4.1, we give a general description of the target. Then, in sec. 4.2, we describe two sublayers of the layer of target's representation in a computer, each corresponding to a separate mechanism of the simulation. The sublayer of the process-level simuation of the internal behavior of the nodes, described in sect. 4.2.1, creates "chunks" or "tasks" of serial computations. These "tasks" present "building blocks" for the higher hierarchy sublayer, comprising the mechanism of the simulation of the external behavior of the nodes, as described in sect. 4.2.2. The latter is designed so as to facilitate the subsequent parallelization of the thus obtained serial TD simulator.

Note that while these two mechanisms are combined in our simulation, properly speaking they are independent. Table-driven mechanism, simulating the process-level behavior of a node, does not assume any particular mechanism of the simulation of external behavior, while the latter interfaces with the former one only by handling the transactions and control-messages generated by it without any knowledge of how these messages were created.

4.1. Target

4.1.1. The configuration of the target

The target system, shown in Fig. 2, represents a service point of a large packet switched communications network. It consists of three front-end processors (terminal concentrators) which act as gateways for the terminals connected to the service point. The front-end processors identified as **FEP1, FEP2,** and **FEP3** in Fig. 2 are **IBM** series/1 machines. The **FEPs** are connected to three node processors, denoted by **CPU1**, **CPU2**, and **CPU3**. These processors, when in need of the database access, communicate with the database processor, **DBP**. The **CPUs** and the **DBP** are **VAX 11/780** hardware with a **VMS 3.0** operating system. The terminals connected to the service point are denoted by the node **TERMINALS**, in Fig. 2. These terminals vary in their speed (300 baud to 9600 baud), mode of access (asynchronous, synchronous, and cluster controlled), and the type of activity.

Thus the target system consists of eight nodes, seven of these nodes represent processors and the other remaining node represents the collection of terminals. In our experiments, we have simulated the target system with the number of terminals varying from 10 to 100.

4.1.2. Entities and endogenous events of the simulated target system

Since the target system was simulated at process level, many entities and endogenous events were included in the simulation.

Entities of the Simulation

(1) Terminals

(2) processors

(3) processes residing in each processor.

(4) process ready and wait queues

(5) The scheduler modules of the operating system.

(6) Process control blocks and system tables.

Items 3) through 5) were included for the 7 processing nodes in the target system (3 **FEPs**, 3 **CPUs**, and 1 **DBP**). The memory management portion of the operating system was not simulated in this initial version of the distributed simulator.

The endogenous events induced by the interaction of the entities above are enumerated below.

Endogenous events:

(1) Arrival of a customer to a process queue

(2) Departure of a customer from a process queue

(3) State changes of a process between ready, current and wait states.

4.2. Serial TD simulation of the target

4.2.1. A table-driven approach to simulating on-line computer networks at process level

4.2.1.1. A general overview of the table-driven approach

This section discusses a new approach to discrete event simulation of computer networks. This approach, called the table-driven approach is embodied in our distributed simulator. The concept of table-driven simulation is independent of the distributed simulation and hence applies as well as to the uniprocessor simulation.

Many current day computer systems are characterized by a network of heterogeneous processors servicing a collection of on-line terminals. The network of computers works asynchronously. Typically, each computer in the network in an "expert" at solving a special class of problems. For instance, one computer can be an "expert" in databases while another computer can be an expert at numerical problems. The design and analysis of such computers poses unique challenges and issues. Synthesis and *detailed* design of these large systems is typically accomplished by a **GPSS** based or C-based discrete event simulation models. These models usually take one to two man-years to develop and debug. The discrete event simulation programs emulate the flow of transactions inside the computer systems and gather statistics on various performance measures of the network, like response times, utilization of processors, etc. The underlying infrastructure of these simulation programs are the same, i.e. emulating the flow of transactions inside the computer network.

The table-driven approach attempts to exploit this commonality of these systems and provides a higher level view of the flow of transactions inside the computer network. In this higher level "view," the transaction flows occurring inside the computer network are specified in various tables. The simulator takes this specification and accomplishes the discrete event simulation of the system. The user is thus relieved of the drudgery of writing and debugging simulation programs.

The primary motivation for the development of a table-driven simulator is to reduce the time and effort involved in developing discrete event simulation models of on-line computer networks. This reduction is accomplished by providing the user with a very high level view of the computer network. This high level view is rooted in the concept of "transactions" flowing inside the computer network. Unlike, queueing networks, however, where the modeling occurs at the resource level, the table-driven simulator also consider the processes, the schedulers and operating systems, in addition to the resources.

The structure of the table-driven simulator is as shown in Fig. 3. First, the user encodes the global connection between processors in the network, in the Global Interconnection Table. This interconnection specifies *physical* links between processors in the network. At the next level, the user encodes the terminal behavior during one session of interaction with the network. The terminal behavior indicates the sequence of commands that will be initiated from the terminal during a session. The terminal behavior is encoded in the script tables. At the next level, the user encodes the transaction flows occurring inside the computer network, in scenario tables. There is the scenario table for each command recognized by the computer network. The information contained in scenario tables indicates the processes involved in the transactions as well as routing within the computer network. As a final specification, the user informs the simulator what operating systems are resident in each hardware of the computer network. The current version of our simulator allows for the **DEC VMS/3.0** operating systems to be resident in each node of the computer network. As shown in Fig. 3, the simulator code is invariant. It parses the global interconnection table, the script tables, and the scenario tables, to completely identify the process structure and the transaction flows inside the computer network. The simulator then emulates the behavior of each terminal by executing the corresponding script table. Each command encoded in the script table invokes the corresponding scenario table for that commend. The simulator executes the scenario table appropriately tracing the transaction flows. Finally, the simulator collects statistics on all relevant performance measures. We will know examine, in detail, the contents of various tables.

4.2.1.2. The components of the table-driven simulator

GLOBAL INTERCONNECTION TABLE (GIT)

GIT encodes the physical interconnection of processors in the computer network by enumerating the adjacency list of each node and its reachable neighbors.

An example of a GIT

i) Consider the computer network shown in Fig. 4. Assuming that each line in Fig. 4 connecting the nodes represents a bidirectional link, the Global Interconnection Table would be:

ter	:	cpu1, cpu2
cpu1	:	disk1, disk2, ter
cpu2	:	disk1, disk2, ter
disk1	:	cpu1, cpu2
disk2	:	cpu1, cpu2

SCRIPT TABLES (ST)

STs describe the behavior of terminals interacting with the computer network. The behavior of terminals during a session with the computer network depends upon the task that the person using the terminal wishes to accomplish. Thus the user may supply several STs. For instance, if the task corresponds to *electronic mail*, then the commands that will be initialed from the terminal will be: *logon*, *edit message*, and *mail message*. If the task corresponds to *compiling* and *running*, then the series of commands initialed from the terminal will be: *logon*, *compile*, and *load & go*. Each terminal that is simulated is assigned one of the STs during a simulation run. The state descriptor of the terminal has a ST *id field*, and a ST *pointer*. The ST *pointer* always points to the row in the script table, that the terminal is currently executing. At the commencement of the simulation, all ST *pointers* in the state-descriptors of terminals are initialized to zero. The terminals advance their state by executing the command in each row of the script table, until the last row of the script table is executed. Immediately following this, the ST pointer wraps-around and begins execution of the first row, and the whole cycle is repeated.

The structure of the script tables will now be described. In this initial version of the TD simulator, there are exactly three columns in the ST. The first column contains the command, indicated symbolically; for example;

logon, Survey-message, etc. Any command in the command repertoire of the computer network is a valid entry in this column. The second column contains the iteration count. This count denotes the number of times the command in that row will be executed, before advancing to the next row. Hence this feature represents a convenient way of specifying a tight loop consisting of one command. A special command, known as the "loop" command is provided by the table-driven simulator for executing larger loops. The "loop" command is a special command recognized only by the table-driven simulator (it is not part of the command repertoire of the computer network being simulated). The first column of the row containing the loop command, has the symbol "loop." The second column contains the loop iteration count. The third column contains the *row* index to transfer to, if the loop has not been executed iteration-count times. If the loop has been executed iteration-count times, then the control transfers to the next row. Loop commands can be nested to any depth to model more complex terminal behavior.

For normal commands specified in the script table, the third column represents the mean time the user thinks (and types) before initiating that command. The variability in think times and typing speeds between novice and expert users is simulated by assuming that the think times are exponentially distributed with the given mean.

Given below are some examples of script tables.

Example of a script table to perform Electronic Mail.

The script table shown in Fig. 5 captures the terminal activity of a person sending an electronic mail consisting of 20 lines. The user initially logs on to the system after thinking for 5 secs. He then invokes the text editor using the "edit" command. After interacting with the editor 20 times, the message is written out using the "write" command. Subsequently the "mail" command is invoked to "mail" the message. Finally the user logs off the system using the logoff command. The "nop" command in the script table is a special command recognized by the simulator as a "do nothing" command. The "nop" command is generally used to simulate the terminal idle status. Hence in this example, the terminal idles for 300 seconds before the whole script table is reexecuted.

SCENARIO TABLES

Scenario tables describe the transaction flows occurring inside the computer network as a result of the command being initiated from the terminal. There is one scenario table for each command invokable by the terminal. The contents of the scenario tables contextually define the processors, processes inside the processor, and the conditional routing probabilities. An example shown in Fig. 6 will elucidate the structure of scenario tables. In scenario tables, the first column denotes the processor name and the second column denotes the process name. The scenario table shown in Fig. 6 corresponds to the transactions flows occurring inside the computer network of Fig. 2 (the target system), for the "logon" command. When "logon" command is executed, the transaction initially joins the queue of **TERMINAL PROCESS** inside the processor **FEP**. Here both the processor name **FEP** and the process name **TERMINAL PROCESS** are generic. (All symbols in Fig. 6 shown in upper case are generic.) As shown in Fig. 2, there are three **FEP**'s in the computer network. Each terminal executing the "logon" command will consult its terminal descriptor data structure, to determine which **FEP (FEP1, FEP2, or FEP3)** this terminal is hardwired to, and will join the queue of that processor. Similarly the generic process name **TERMINAL PROCESS** will be translated to the process name of the terminal process contained in the terminal descriptor data structure. The third column in the scenario table represents the mean service time[2] required by the transaction in the process. Thus in the example of Fig. 6, the logon transaction will require, on the average, 100 milliseconds of **CPU** time from the "terminal process." Since the "terminal process" itself will contend with other processes in the **FEP**, this transaction may suffer a delay in the "terminal process" considerably longer than the 100 milliseconds. There are two components in this delay:

i) Congestion occurring in the **TERMINAL PROCESS**.

ii) Scheduling of the **TERMINAL PROCESS**, based on its priority relative to other processes. **TERMINAL PROCESS** may not be scheduled to acquire the **CPU** for some time, and while possessing the **CPU**, may be preempted by a higher priority process.

Columns 4 to 9 of the scenario table are reserved for conditional branching. These columns are treated in pairs. The first element in the pair denotes the probability of transferring to the row index given in the second element of the pair. Thus after executing any given row of the scenario table, the transaction will perform a coin toss, and probabilistically branch to one of the alternative processes. If no entries exist in columns 4-9, the transaction will unconditionally proceed to the next row of the scenario table. Now continuing on with the example, the logon transaction, after completing service at the terminal process will unconditionally join the npi process in the appropriate **FEP** processor. Npi stands for "Node Processor Interface" and is the process that interfaces with the node processors **NP** as shown in Fig. 2. After obtaining 200 milliseconds of service from npi, the logon transaction proceeds to the node processor **NP** to join the process fepi. Here, **NP** is again a generic processor name. The actual processor the logon transaction joins will again be determined by the information contained in the terminal descriptor data structure. After obtaining service from fepi, the logon

[2]All service times in the scenario tables are assumed to be exponentially distributed.

transaction joins the queue of the process lap, the Logon Application Process. This process validates the user identification information which takes 50 milliseconds. As shown in Fig. 6 in row 4, a coin toss is now made to determine the next process to join. This coin toss models the real life phenomenon that the user identification is invalid because of typing errors, about one every 1000 times. Thus, with a probabilities of .001 the transaction proceeds to row 11 and with a remaining probability of .999, it proceeds to the next row (Row 5). The transaction that is in error, then returns back to the terminal via the processes fepi, npi, and **TERMINAL PROCESS**. The transaction whose user id. has been validated proceeds on to the **DBP**, possibly to download the user profiles, etc. The valid transaction, after being processed at the **DBP**, returns back to the **NP**, and finally to the terminal.

To summarize, scenario tables model the detailed transaction flows inside the computer network, the flow resulting from the initiation of commands from the terminal. Each command, potentially invokable from the terminal, has a corresponding scenario table. The optional arguments in commands, small variations in the command, and typing errors in the command are modeled using probabilistic branching of transactions in the scenario tables. Several other important phenomena, like transaction spawning, dynamic creation of processes, etc. are not modeled in the initial version of the table-driven simulator.

OPERATING SYSTEM MODULES

The processes that are defined in the scenario table have an implicit priority as well as scheduling policy associated with them. As a final level of information, the user supplies the priorities of processes, the scheduling philosophy, and the time quantum associated with each priority level. In the target system simulated, all the node processors (**NPs**) and the data base processor **DBp** were running **VMS/3.0** operating system. The **FEPs** were running **SERIES/1** operating system. Hence the simulator currently has a version of **VMS/3.0** operating system and a version of **SERIES/1** operating system defined. However, the modular structure of the simulator, as shown in Fig. 3, facilitates incorporation any user supplied operating system(s) into the simulator. In the **VMS/3.0** operating system currently incorporated into the simulator, the memory management schemes are left out. Thus the simulator assumes that all processes are resident in main memory at all times, and are never swapped out.

In conclusion, the table-driven simulator provides a simple high level view of the on-line computer network, in terms of scripts, scenarios, and operating system interactions. The invariant simulator code can thus simulate many computer networks with the loading of appropriate tables. We found that in going from the target system shown in Fig. 2, to a larger system with 14 nodes, the reloading of tables took about a week, and the 14 node simulator was up and running. With the incorporation of many standard operating system modules into the simulator, the table-driven simulator may find wide use.

4.2.2. Simulation of the external behavior of the nodes

4.2.2.1. A general overview of the external simulation

The following simplified code outlines the serial simulation of the external behavior of the nodes (without initialization and termination structures):

```
cycle:
     advance global clock                          (4.1)
     for all nodes do CHN (node)                   (4.2)
     for all nodes do ADN (node)                   (4.3)
```

The simulated time in (4.1) is advanced in steps. A step of the time advancement is computed in two phases. At the *check−node* phase (4.2) subroutine CHN decides which of the incoming messages (if any) are accepted by the node, then at the *advance−node* phase (4.3) subroutine ADN advances the history one or more steps ahead. More detailed description follows.

In order to pass transactions and control-messages between the nodes we introduce *mailboxes*. A mailbox is an integer variable associated with a directed link in the network graph. Value 0 of a mailbox encodes the status of the absence of a message, values 1, 2,... encode both the presence of a message and the message *type*.

Being at large a TD algorithm, an ED "flavor" is introduced which enables time to be advanced more than one time step ahead, if no one node has anything to do at these steps, as explained later in sec. 4.2.2.3.

4.2.2.2. The simulation of the node communication

SIMPLIFIED CASE: We first consider the case of a two-node network, in which the *sender* node is connected with the *receiver* node via the mailbox m and a unidirectional link. Initially $m = 0$. An individual communication act is a result of computations at two consecutive time steps, let they be $n-1$ and n. Before the end of the time step $n-1$ during the ADN phase, the sender, based on its local state decides whether or not it attempts to send a message to the receiver, and what is the type of the message, if any. when the attempt is made, the sender writes the corresponding positive value into location m. At time step n at the CHN phase, the receiver reads the value of the mailbox m, and based on its local state, decides whether or not it accepts the corresponding message if any. If the message is rejected then the contents of the location m remain unchanged otherwise the receiver writes 0 into m. At the beginning of the phase ADN at step n, the sender reads m if it was attempting to send a message at step $n-1$. A value $m > 0$ ($m = 0$) is then the indication of the non-acceptance (acceptance) of the message. At the end of the ADN phase the sender may retry sending the same message if it was rejected, send another message, or do not send any message at all. In the latter case the value 0 must be written into m by the sender, since if the sender does not write the mailbox which previously was non-zero, then the receiver will treat the mailbox contents as the indication of the presence of a message (retrying).

THE GENERAL CASE: We now extend the above description on to a general network communication graph. A node in such a graph has a number of *input* and *output* links. Each link is associted with a pair (*sender node*, *receiver node*) and hence with exactly one mailbox. A node may act as both a sender and a receiver. Ending the ADN phase at cycle $n-1$ a node may or may not write one or more of its output mailboxes (i.e. those associated with its output links). This represents sending message(s) at the and of simulated cycle $n-1$. Starting its CHN phase at cycle n the same node checks all of its input mailboxes (i.e. those associated with its input links). It then takes into account its own state and status of its input mailboxes and decides which of incoming messages it accepts (if any), the others being rejected. Then, at the end of the CHN phase, the incoming mailboxes are marked accordingly (value 0 is written or they remain unchanged). We close this description by mentioning that at the beginning of the phase ADN at cycle n the node reads its output mailboxes verifying acceptance of the messages sent by it at the end of cycle $n-1$. Fig. 7 depicts the handshake protocol described above.

ADEQUACY OF THE PROTOCOL: We believe that the suggested simple protocol truthfully represents the interprocessor communication when only timing considerations are of importance, and when transmission errors are ignored.

Let us discuss the correspondence between the events in a real computer network when a message is being sent and the representation of these events in our protocol. A computer node A at the time interval $((n-1)\Delta, n\Delta)$ decides to send a message to a computer node B. In the case of the message acceptance the physical message transmission and digestion including error correction has taken place in the interval of length $\delta \ll \Delta$, and terminates no later than time $n\Delta+\delta$. Almost the whole slice $(n\Delta, (n+1)\Delta)$ is then at the disposal of node B to react accordingly to the message received (e.g. to queue up a request, to preempt, to start processing from the idle state etc.). In our model the message is considered to be under processing since time $n\Delta$.

On the other hand if the message is rejected, then physically node A may react on this event starting with the very moment of rejection which may be one of the interval Δ(n-1),Δn , whereas in our model node *A* gets notice of its message rejection slightly later, only at the time slice *n*, i.e. at the interval (Δ*n*, Δ(*n*+1)). However in all the software structures known to the authors the communication protocol provides a reaction on a message rejection no earlier than at the next time slice. In particular there is no reason for an earlier (than the start of the time slice *n*) retry for sending the same message since nothing has been changed in the status of the receiver with respect to the acceptance of the given message.

Thus both these possibilities can be adequately represented in terms of the suggested protocol.

4.2.2.3. Skipping blank computational steps

To increase the efficiency of the simulation, we have designed a mechanism for skipping the time periods when the whole network is inactive. To this effect, subroutine ADN returns to the external sublayer the value `tearl` of the estimated time of the earliest event, expected by the given node. The external sublayer, in turn, computes the next time step as minimum of `tearl`s over all the nodes. The current global time is kept in the variable `tglob`, and the computed next time step is kept in the variable `next`. We observe that always `next` ≥ `tglob` + 1. If `next` > `tglob` + 1, then `next - tglob - 1` blank steps are skipped.

5. Adapting serial algorithm for parallel simulation

5.1. Parallelization of cycles

Both cycles (4.2) and (4.3) can be programmed in FORTRAN as follow

```
      do 10 instance = 1, multiplicity
         call  TASKNAME (instance)         }          (5.1)
   10 continue
```

In cycle (5.1) `instance` is the cycle parameter, `multiplicity` is the number of instances of a task `TASKNAME`. The latter is either CHN in (4.2) or ADN in (4.3). Assuming that all instances of these tasks are independent from one another, which would be a case if each of them may modify only its own set of variables, we parallelize cycle (5.1) by using the following code in FORTRAN.

```
      call SYNC&SET(multiplicity)                (5.2)
      while (SATREQ (instance)) do               (5.3)
         call TASKNAME (instance)                (5.4)
      end while
```

`Subroutine SYNC&SET` traps all PEs until all of them complete their previous computation and then releases PEs. As a side effect before this release this subroutine initializes MULTIPLICITY and initial value for a variable that is used in the logical function SATREQ. An invocation of SATREQ by a PE either returns TRUE and, as a side effect, a value of private instance in the range from 1 to multiplicity or returns FALSE, value of instance being unused in this case, if all instances of the task are already taken for execution. Note that the value of instance is unique for each invocation of SATREQ by each PE, so no one instance of task would be duplicated.

Codes for subroutine SYNC&SET and function SATREQ are given in Fig. 8. In these codes we replace FORTRAN assignment symbol "=" by "<-" and comparison predicate symbols ".lt." and ".gt." by "<" and ">", respectively. Shared variable dis-count contains the value of task instance to be distributed next.

```
shared: limit, dis-count, #free (2)
private: index, instance
constant: total-#-pes
```

```
subroutine SYNC&SET (multiplicity)

data index /1/                                        (5.5)

if (F&A (#free (index), 1) < total-#-pes - 1)  (5.6)
then go to 10
   dis-count <- 1                                     (5.7)
   limit <- multiplicity                              (5.8)
   #free-pes (index) <- 0                             (5.9)
   go to 20                                           (5.10)

10     if (#free-pes (index) > 0) then go to 10    (5.11)

20   index <- 3 - index                               (5.12)
     return
```

```
logical function SATREQ (instance)

SATREQ <- TRUE                                        (5.13)
instance <- F&A (dis-count, 1)                        (5.14)
if (instance > limit) then SATREQ <- FALSE     (5.15)
return
```

Fig. 8. SYNC&SET and SATREQ coordination primitives.

5.2. Ensuring reproducibility of the simulation

Given fixed initial conditions one naturally expects to obtain the same history of target simulation independently of the host or of the local condition for a particular run. Being a problem in a uniprocessor environment, this requirement attains an additional flavor in a parallel environment, since the order in which the parallel chunks of computation are interleaved may depend from host to host and from a run to a run. Thus, all "parallel" reasons of possible irreproducibility would be eliminated if the results of the computation don't depend on the order of interleaving.

In our program the order of interleaving might become critical in the following two ways:
(i) when several PEs execute floating-point *F&A* (V,e) over the same shared location V and the truncation scheme violates the associative law of addition. For example, final value of V computed as $V = V'+e_3$ with V' computed as $V' = e_1+e_2$ may be not equal to the V computed as $V = e_1+V''$ with V'' computed as $V'' = e_2+e_3$.
(ii) when the random number sequence a task uses depends on the order of interleaving. For example node **TERMINALS** at a given simulated instance t may use different random numbers depending on the PE which is currently carrying the corresponding task.

Our algorithm does not use floating-point F&A. This operation is only used for coordination purposes, all the *F&A*s having *integer* arguments (see codes in Fig. 8). For integer numbers the associative laws are not violated by computers, since no truncation is involved. Thus (i) may not be a reason for irreproducibility.

To eliminate (ii) in our simulation the following technique was used. An array `rseed` of random number seeds was allocated in shared memory, its dimension being the number of nodes in the target, and the initial values of the seeds being the input information. Each time a task on behalf of a node starts executing on a PE, this PE reads `rseed(node)` and stores the value into its private location `rand`. During the task execution this `rand` is used in the usual recursive manner by a random number generator when new random numbers are needed. When the task execution is completed the final value of `rand` is stored back into `rseed(node)` thus providing reproducible succession of random numbers used by a `node` independently of the PEs which executed tasks on behalf of this node. (Of course, all PEs use copies of the same random number algorithm.)

5.3. Overall structure of the task manager in the host

In Fig. 9 the items in capital letters are: subroutines CHN and ADN, coordination primitives SYNC&SET and SATREQ, presented in Fig. 8, P and V semaphore subroutines (whose *F&A* implementation one may find in [6]), and subroutine FINISH which provides a special handling for parallel termination, the details of this subroutine are omitted here, since they are very much environment-dependent. (The FINISH we used is compatible with the environment of the program that simulates host on the host simulator). Note that exclusive write to mailboxes is ensured by the synchronization mechanism, since no CHN-phase can interleave with an ADN-phase.

```
shared: tglob, rseed, sem
private: node, rand, tearl
constant: very-large-number, last-cycle, node#
initially tnext = very-large-number, tglob = 0
```

```
10   call SYNC&SET (node#)                                   (5.16)
      if (tglob > last-cycle) then call FINISH
      while (SATREQ (node)) do
         rand = rseed (node)
         call CHN (node,rand)
         rseed (node) = rand
      end while                                              (5.17)

      call SYNC&SET(node#)                                   (5.18)
      while (SATREQ (node)) do
         rand = rseed (node)
         call ADN (node, rand, tearl)
         rseed (node) = rand

         call P (sem)                                        (5.19)
            if (tearl < tnext) then
                tnext = tearl
            end if
         call V (sem)                                        (5.20)
      end while                                              (5.21)

      call SYNC&SET (1)                                      (5.22)
      while (SATREQ (node)) do
         if (tnext < large-number) then
            tglob = tnext
            tnext = very-large-number
         else
            tglob = very-large-number
         end if
      end while                                              (5.23)

      go to 10
```

Fig. 9. Main routine of the simulator (task manager).

6. A remark on the mechanism of the simulation of the host

Thus far the discussion has been focussed on the two layers of our simulation, target and host, and their relationships. Below we discuss the third layer, host-simulator, and its relationship to the host.

The host, the Ultracomputer, is simulated on the host-simulator (CDC-6600, CYBER, NYU-PUMA) by the program called *Washcloth* [4]. The input to *Washcloth* is the load module of the assembly-language code of a parallel program. This code is simulated in a TD fashion: in its regular mode, *Washcloth* assumes[3] that each PE spends one machine cycle for each machine-language instruction; a global clock, an integer counter of elapsed simulated cycles, is being maintained; for each value of this counter, *Washcloth* scans each PE and advances the PE's state one cycle ahead.

To generate the input to *Washcloth*, we compile the FORTRAN code of our simulator using a regular compiler, which, naturally, does not distinguish our parallel program from an ordinary serial program. During the compilation all the non-through-*F&A* references to the shared memory are reduced to the usual loads and stores. *F&A* references are treated by the compiler as ordinary external function references. During simulation, *Washcloth* handles the shared memory references differently from the private references. (Using FORTRAN conventions, a method is provided to indicate to *Washcloth* whether a variable is shared or private.)

7. Experimental results of the parallel simulation

The aim of the simulation experiments was to study how the efficiency of the parallel algorithm depends on the size of the parallel host computer and the size of the problem. Due to the memory/speed limitations of the host-simulator, we could only run problems of relatively small sizes of target network and parallel host. We therefore tried to use these results to make a reasonable prediction for larger target networks and a larger host.

7.1. Nomenclature Recall the commonly accepted definitions of efficiency and speed-up. The ideal efficiency $E_{ideal}(P)$ of an algorithm executed by a parallel computer consisting of P PEs is given by

$$E_{ideal}(P) = \frac{Ts_{best}}{PTp(P)} \tag{7.1}$$

where Ts_{best} is the time required by the best possible serial algorithm, and $Tp(P)$ is the time spent by the parallel algorithm with P PEs. The speed-up is the efficiency $\times P$.

[3]*Washcloth* has also several irregular modes. In one irregular mode, for example, speeds of executions by different PEs are different.

Since the best serial algorithm is usually not known, Ts_{best} above is replaced by its estimate Ts, and (7.1) may be rewritten as

$$E_{ideal}(P) = \frac{Ts}{Tp(1)} \frac{Tp(1)}{P\,Tp(P)}, \tag{7.2}$$

where the ratio $e_0 \equiv \frac{Ts}{Tp(1)}$ measures the loss due to the serial code being modified to make it parallel and the ratio $e(P) \equiv \frac{Tp(1)}{P\,Tp(P)}$ measures the loss due to the synchronization involved in the execution of the parallel code by P PEs.

7.2. Numerical results of the simulation

The basic experiment was with the target network consisting of 8 nodes (Fig. 2) node 1 representing 50 terminals. A particular history of target activity of the order of 3000 seconds of target time was chosen. After an initial period of about 300 secs. of transient behavior, the target came to its stationary regime and host time measurement (in the host cycles) was started and kept till the end of the history. Table 1 presents speed-ups and efficiencies of the parallel simulation for various P.

P	1	2	3	4	5	6	7	8
speed-up	assumed 1	1.80	2.41	2.94	3.55	3.95	4.18	4.20
$e(P)\%$	assumed 100	90	80	74	71	66	60	52

Table 1. Performance for an '8 nodes, 50 terminals' experiment.

As one might expect, speed-up increases with the increasing P while efficiency decreases. Since 8 is the maximal number of concurrent tasks in this example, values of P greater than 8 were not tried ($P-8$ PEs would be always idle for such cases, speed-up would remain the same, while efficiency would decrease as $e(8)\frac{8}{P}$). The maximal attainable speed-up, 4.20, is almost two times less than its theoretical upper bound, 8, due to synchronization penalties.

7.3. Synchronization penalties

A task may be *empty*, when no event occurs in the node, and *non−empty* when a new event is simulated. The empty tasks are the penalty for the TD character of our simulation; in an ED simulation such tasks never occur. Fig. 10 and 11 depict an initial segment of one simulated history of the '8 nodes, 50 terminals' target by a 3 PEs' parallel host. Note that since the tasks corresponding to the CHN phase are 2 orders of magnitude smaller than the non-empty ADN tasks, the former are ignored in these depictions as if each grand loop "10...go to 10" in Fig. 9 would contain a single instance of task distribution (in reality this loop contains three such instances). Fig. 10

thus presents a typical activity pattern where crosses represent the non-empty tasks. For example at time 0 only node 1, **TERMINALS**, is active. A terminal probably generates a message to be sent into the network. While node 1 is still active at time 1, node 2, a **FEP**, is also active. This **FEP** is probably accepting the message generated at time 1, etc. Since $P = 3$ is less than number of nodes in the network, a substantial task reallocation is taking place. For example, PE 1 starts executing task on behalf of node 1 for the simulated time 0, while the two other PEs execute the other 7 small empty tasks.

7.4. An experiment with a modified scheduling

Observe that the order of the scheduling is not predictable in full detail, and, generally, we don't know which task will be executed by which PE. However, since our program (see Fig. 8 and 9) queues the tasks according to the order number of the corresponding nodes, a certain restriction on this scheduling is imposed. Thus **TERMINALS** are always taken for execution first, then 3 **FEPs** follow in the predetermined order, then three **CPUs** and finally the **DBP**. Since the execution times of the chunks corresponding to the different kinds of tasks may be different (for example **TERMINALS** tasks are always much smaller than any other tasks, while **DBP** tasks are usually the largest), we tried to improve performance by redefining the order on which the tasks are queued.

In a variation of our basic '8 nodes, 50 terminals' experiment the positions of the **DBP** and **TERMINALS** tasks in the queueing-up order were interchanged. With this interchange one is not to expect a gain for $P = 1$ or $P = 8$. For the intermediate values of P, the maximum improvement in the efficiency observed was only about 1%, and this maximum was achieved for the value of $P = 3$ *or* 4.

7.5. Prediction of performance for larger size problems

Predicting efficiency in the simulation of a network of arbitrary size on a parallel computer of arbitrary size is difficult for two reasons:
(i) The mutual distribution of concurrent task durations for the same instance of task splitting as well as for different time instances is very complicated. In fact *a posteriori* chosen algorithm of functioning the network can invalidate any *a priori* assumptions concerning such a distribution.
(ii) When number P of PEs is less than the number of tasks (= the number of nodes in the network), the rule of assigning a PE for a task complicates the analysis.

There are not very many techniques that have been developed for analyzing efficiency, and none of the known techniques answer our question. For example the method in [10] gives a theoretical analysis of the case where there are n independent tasks with equally distributed exponential durations and n PEs are ready to execute these tasks. Note than if n increases while the distribution remains the same, then efficiency tends to zero due to the existence of rare tasks that are very lengthy. This follows from the formulas in [10].

The exponential assumption turns out to be very unrealistic. In fact, task length is limited. We call this limit X. Below, we suggest a very simple *lower bound* for the efficiency with the following two assumptions:

(A1) X is known and does not depend on the size of the network.

(A2) There exists and is known a positive lower bound x of the mean μ of the distribution of a task duration and this x does not depend on the size of the network.

While (A1) may be assured by finite buffers etc., one may easily argue that (A2) is doubtful. We may advocate (A2) by mentioning that in the simulation only the maximally loaded systems are considered. We don't see why, under this condition of full load, durations of task executions for simulating larger and larger networks should become smaller and smaller. (This is essentially what the negation of (A2) means.)

Consider the case of P coinciding with the number of nodes in the network. Then assumptions (A1) and (A2) give raise to the following simple lower-bound estimate of efficiency:

$$e(P) \geq \frac{x}{X}. \tag{7.3}$$

We will show that estimate (7.3) is valid only asymptotically when the number of tasks executed increases, and when the empirical mean μ_{emp} is close to the ideal mean μ.

We denote by X_{emp} the empirical estimate for X. If M tasks have been executed, the length of the task i being L_i, then $X_{emp} = \max\limits_{i=1,\ldots M} L_i$.

Proposition. If the operational time to schedule the tasks is negligible compared to the lengths of the tasks and each task splitting instance contains exactly P tasks, then

$$e(P) \geq \frac{\mu_{emp}}{X_{emp}}. \tag{7.4}$$

Estimate (7.4) holds no matter in which order tasks are taken for execution and what P is.

Proof. We denote by S the set of all M tasks executed, $S = \{1,\ldots M\}$. The tasks of S are partitioned into k instances: $S = \bigcup\limits_{j=1}^{k} S_j$, $S_j \bigcap S_r = \varnothing$, $j \neq r$, $1 \leq j,r \leq k$, so that $M = kP$ and at instance j, $1 \leq j \leq k$, all P tasks of the subset S_j are executed in parallel by P PEs.

We assume that for each j all P PEs start executing the tasks of S_j simultaneously and no one of them is released until all the tasks are completed. Therefore the work (number of instructions expended by the busy PEs) to execute all tasks of S_j is $P \times \max_{i \in S_j} L_i$, while the useful work done is $\sum_{i \in S_j} L_i$. Totaling the expended and the useful work for all k instances and relating the latter to the former, we obtain

$$e(P) = \frac{\sum_{i=1}^{M} L_i}{P \times \sum_{j=1}^{k} \max_{i \in S_j} L_i} \leq \frac{\sum_{i=1}^{M} L_i}{M \times \max_{i=1,\ldots M} L_i} = \frac{\mu_{emp}}{X_{emp}}.$$

End of proof.

Under usual assumptions of the ergodicity, right-hand side of (7.4) may be replaced by the right-hand side of (7.3) when the time of simulation increases. It is in this sense that we claimed (7.3) to hold only asymptotically. In practice, we don't distinguish between (7.3) and (7.4).

Note. It is well-known that $P' < P$ does not necessarily imply $e(P) \geq e(P')$, although this implication always takes place in our experiments (e.g. see the efficiency figures in Table 1). For example, if $M = P = 3$, $k = 1$, all the three tasks being of equal length, then $e(3) = 100\%$ (all the three PEs start and finish simultaneously), while $e(2) = 75\%$ (the two PEs start two of these three tasks and finish them simultaneously, then the remaining task is executed by one PE, the other being idle). Generally neither (7.3) nor (7.4) holds if P is smaller than the number of nodes in the network. In this case the inequalities $e(P) \geq \frac{x}{2X}$ and $e(P) \geq \frac{\mu_{emp}}{2X_{emp}}$ hold instead of (7.3) and (7.4), respectively.

In our basic experiment, time measurements give value 28% for the right hand-side of (7.4). Thus, assuming that right-hand side of (7.3) is also about 28%, we predict that the efficiency for larger sizes network of the similar structure will not fall below about 28%. One experiment with the a larger network (100 terminals, 6 **FEP**s, 6 **CPU**s and 1 **DBP**; total of 14 nodes) showed efficiency not smaller than 56% (speed-up 8.1 for 14 PEs) while its low bound estimate as in (7.4) was 25%.

Thus far we have discussed only the factor $e(P)$ of the efficiency. As to the factor e_0, its expert estimation for our algorithm gives the value[4] about 50%. Measuring the penalty for modification of the serial ode to make it

[4]The "experts" are ourselves; t.e. the authors believe that the presented parallel algorithm is about two times slower when executed by a single PE of the parallel machine, compared with a good serial algorithm performing the same task. No other expert estimation

parallel, the e_0 does not depend critically on the running problem, in particular, on the size of the target network, provided the density of concurrent non-empty tasks is high.

8. A remark on the 'low density of events' case

Property (II) postulated in the introduction is essential for our simulator, in particular, for the suggested way of task scheduling, where *all nodes* are scanned in order to determine the non-empty tasks. For a large network (II) might not be true, i.e. only a small fraction of nodes at a typical time step generates events. Then the large overhead due to the execution of very many empty tasks would make the simulator inefficient. However, if the number of concurrent events is still large, their fraction being small, a more elaborate technique of node scanning is possible. This might require the use of an efficient queue management technique such as the one in [6], and the use of additional data structures.

We did not commit ourselves to such complications, since the large network necessary to demonstrate the advantages of these optimizations can not be simulated on the existing serial host simulators.

9. Conclusion

This paper has demonstrated the usefulness of the parallel TD simulation in obtaining meaningful speed-ups for a class of computer networks that possess inherent synchronism. This paper has also introduced a new table-driven mechanism for simulating computer networks at process level. Our simulator has the potential of becoming a general purpose tool for simulating a large class of realistic computer networks. It is hoped that our results will stimulate interest in parallel TD simulation and generate discussion on the relative merits of it in comparison to parallel ED simulation. It is also hoped that the network we have simulated will become a benchmark for comparison of future parallel TD simulation techniques.

As a continuation on of the work reported here, the authors plan to conduct a parallel ED simulation of a class of computer networks, the results of which will be reported in due time.

10. Acknowledgement

The authors wish to acknowledge the contributions of Debasis Mitra in all aspects of the project. In particular, Debasis Mitra was invaluable in assisting us in constructing the model for speed-up and efficiency prediction.

---

was available: nobody was willing to unravel all the details of our algorithm.

11. References

1. Chandy, K.M., and Misra, J.: "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Comm.ACM, v.24,4, April 1981.

2. Deminet, J.: "Experience with Multiprocessor Algorithms", IEEE Trans. Comp., v.C-31,4, April 1982.

3. General Purpose Simulation System (GPSS), IBM Program Product, Program Numbers 5734-XS2(OS), 5736-XS3(DOS).

4. Gottlieb, A.: "Washcloth - The Logical Successor to Soapsuds", Ultracomputer Note No.12, Courant Institute, NYU, October 1980.

5. Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe,K.P., Rudolph, L., and Snir, M.: "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", IEEE Trans. Comp., Vol. C-32, No. 2, February 1983.

6. Gottlieb, A., Lubachevsky, B.D., and Rudolph, L.: "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," ACM Trans. on Comp. Lang. Sys., Vol.5, No.2, April 1983, pp. 164-189.

7. Heterogeneous Element Processor (HEP), Denelcor, 14221, E. Fourth Ave., Aurora, Colorado, 80011.

8. Jefferson, D.: "Virtual Time", Technical Report TR-83-213, Computer Science Department, Univ. of Southern California, Los Angeles, CA 90089-0782.

9. Kiviat, P.J., Villanueva, R., and Markowitz, H.M.: "SIMSCRIPT II.5 Programming Language," Consolidated Analysis Centers Inc., 1968.

10. Kung, H.T.: "Synchronized and asynchronous parallel algorithms for Multiprocessors," in Algorithms and complexity- New Directions and Recent Results, J.F. Traub, Ed. New York: Academic, 1976, pp. 153-200.

11. Peacock, J.K., Wong, J.W., and Manning, E.: "Distributed Simulation Using a Network of Microcomputers," Computer Networks,v.3, n.1, February 1979.

12. Schwartz, J.T.: "Ultracomputers," ACM TOPLAS, 1980, pp. 484-521.